Kapitel 1 Bits

1.1 Signed og unsigned tal

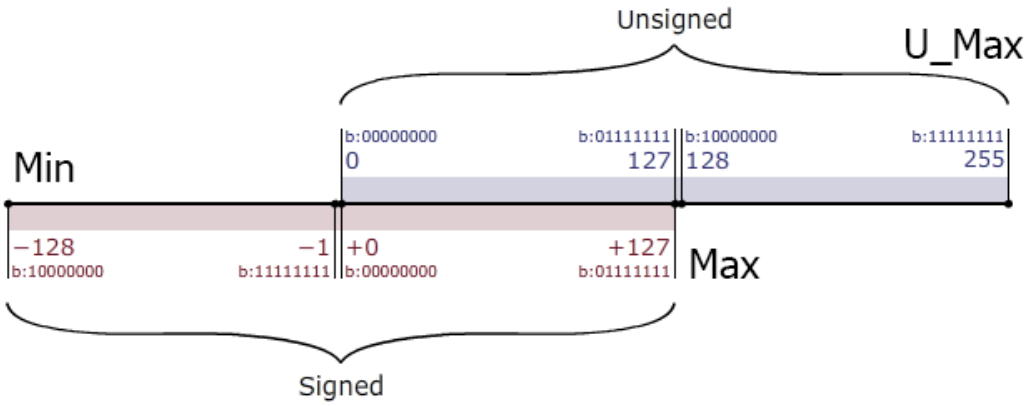
Når en unsigned værdi konverteres til en signed værdi, er most significant bit, den bit der bestemmer om tallet er negativt eller positivt. Denne bit har værdien -8, når den er signed, og derved subtraheres der for alle yderligere bits.

Bits	Unsigned	Signed	Hex
0000	0	0	0x0
0001	1	1	0x1
0010	2	2	0x2
0011	3	2	0x3
0100	4	4	0x4
0101	5	5	0x5
0110	6	6	0x6
0111	7	7	0x7
1000	8	-8	0x8
1001	9	-7	0x9
1010	10	-6	0xA
1011	11	-5	0xB
1100	12	-4	0xC
1101	13	-3	0xD
1110	14	-2	0xE
1111	15	-1	0xF

Tabel 1.1: Tabel over signed og unsigned værdier

1.2 Operationer

For bits findes der operationer, som vist i DTG. Disse er bitvise, og noteres som vist i Tabel 1.2. For Aritmetiske skift, indsættes 1 på den nye plads, hvor ved Logiske skift, indsættes 0.



Figur 1.1: Signed og Unsigned komparativt

Operation	Navn	resultat
x|y	eller	1011 Da disse bit findes i enten eller af strengene.
x&y	og	0001 Da denne bit deles af de to strenge.
x^y	XOR	1010 Da disse bits enten er i den ene eller den anden streng.
~x	negation	1100 Da alle bits vendes.
y»3	Logisk højreskift	0001 Da bitværdien skifter 3 pladser mod højre.
x«3	Logisk venstreskift	1000 Da bitværdien skifter 3 pladser mod venstre.
y»2	Aritmetisk højreskift	1110 Da bitværdien skifter 3 pladser mod højre.
x«2	Aritmetisk venstreskift	1111 Da bitværdien skifter 3 pladser mod venstre.

Tabel 1.2: Bitvise operationer og deres resultater. Der er taget udgangspunkt i $x = 0011$, $y = 1001$

Kapitel 2 Assembly

Assembly sproget er det tætteste vi kommer på maskinkode. Derfor er det essentielt for computer arkitektur faget, at undervise i dette.

2.1 x64 Assembly

2.1.1 Registre

x64 Assembly kode har 16 64-bit registre at gøre godt med. Disse 16 registre kan ydermere opdeles i 32- 16- og 8-bit registre, som vist i Tabel 2.1

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0	reg type
%rax	%eax	%ax	%al	Return value
%rcx	%ecx	%cx	%cl	Callee value
%rdx	%edx	%dx	%dl	4 argument
%rbx	%ebx	%bx	%bl	3 argument
%rsi	%esi	%si	%sil	2 argument
%rdi	%edi	%di	%dil	1 argument
%rsp	%esp	%sp	%spl	Callee saved
%rbp	%ebp	%bp	%bpl	Stack pointer
%r8	%r8d	%r8w	%r8b	5 argument
%r9	%r9d	%r9w	%r9b	6 argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

Tabel 2.1: Oversigt over registre i x64 Assembly

2.1.2 Ord og bytes

Indenfor Assembly arbejdes der med ord og bytes.

- En *byte* er 8 bits (10011011).
- Et *word* er 2 bytes (10011011 10010110).
- Et *dword* er 4 bytes og står for double word.
- Et *qword* er 8 bytes og står for quad word.
- Et *oword* er 16 bytes og står for octoword.

Bogstaverne *d* og *q* indgår også til tider i operationer, som eksempelvis *movq*. Dette betyder blot at der flyttes noget, der er 8 byte stort.

2.1.3 Instruktioner

Mange instruktioner som mov bruger suffix som *w* og *q* når disse bruges. Ydermere findes der flere forskellige former for operationer.

Data flytning

Instruktion	Source/Destination	Beskrivelse
`mov`	S, D	Flytter fra Source til Destination
`push`	S	Skubber til stakken
`pop`	D	Popper toppen af stakken til Destination
`mob`	S, D	Flytter byte til word (Sign)
`push`	S	Flytter byte til word (Zero)
`cwtl`		Konverterer word i `%ax` til dword i `%eax` (Sign)
`cltq`		Konverterer dword i `%eax` til qword i `%rax` (Sign)
`cqto`		Konverterer qword i `%rax` til oword i `%rdx:%rax`

Tabel 2.2: Instruktioner til at flytte data

Aritmetik

Instruktion	Source/Destination	Beskrivelse
`inc`	D	Inkrementerer Destination med 1
`dec`	D	Dekrementerer Destination med 1
`neg`	D	Aritmetisk negation
`not`	D	Bitvis komplement

Tabel 2.3: Unary operationer

Instruktioner som `leq 17(%rax, %rbx, 4), %rcx` skal forstås som `%rcx = 17 + %rax + %rbx * 4`. For operationen `leaq 8(%rax, %rbx, 8)` gælder det, at det sidste 8, er et offset for addressen.

`leaq`	S, D	Indlæser effektiv addresse for Source til Destination
`add`	S, D	Adderer Source til Destination
`sub`	S, D	Subtraherer Source fra Destination
`imul`	S, D	Multiplerer Source med Destination
`xor`	S, D	Bitvis XOR Destination med Source
`or`	S, D	Bitvis OR Destination med Source
`and`	S, D	Bitvis AND Destination med Source

Tabel 2.4: Binære operationer

`sal/shl`	k, D	Venstreskift Destination med k bits
`sar`	k, D	Aritmetisk højreskift Destination med k bits
`shr`	k, D	Logisk højreskift Destination med k bits

Tabel 2.5: Binære operationer

`imulq`	S	Unsigned fuld multiplikation af `%rax` med S. Resultatet gemmes i `%rdx:%rax`
`mulq`	S	Signed fuld multiplikation af `%rax` med S. Resultatet gemmes i `%rdx:%rax`
`idivq`	S	Signed dividering af `%rdx:%rax` med S. Kvotient gemmes i `%rax`. Rest gemmes i `%rdx`
`divq`	S	Unsigned dividering af `%rdx:%rax` med S. Kvotient gemmes i `%rax`. Rest gemmes i `%rdx`

Tabel 2.6: Specielle Aritmetiske operationer

Type	Form	Værdi	Navn
Immediate	`$imm`	imm	Immediate
Register	r_a	R[r_a]	Register
Memory	imm	M[imm]	Absolute
Memory	(r_a)	M[R[r_a]]	Indirect
Memory	$imm(r_b)$	M[$imm + r_b$]	Base+displacement
Memory	(r_b, r_i)	M[R[r_b]+R[r_i]]	Indexed
Memory	$imm(r_i, r_b)$	M[imm + R[r_i]+R[r_b]]	Indexed
Memory	$(, r_i, s)$	M[R[r_i] $\cdot$ s]	Scaled Indexed
Memory	$imm(, r_i, s)$	M[imm+R[r_s] $\cdot$ s]	Scaled Indexed
Memory	(r_i, r_b, s)	M[R[r_i]+R[r_b] $\cdot$ s]	Scaled Indexed
Memory	$imm(r_i, r_b, s)$	M[imm+R[r_i]+R[r_b] $\cdot$ s]	Scaled Indexed

Tabel 2.7: Memory operations

2.2 Y86 Assembly

2.2.1 Process stadier

Processer kræver instruktioner, der kan organiseres som følgende.

Fetch

Her læses bytes for en instruktion, fra hukommelsen. Dette gøres ved at bruge en Program Counter *PC* som hukommelses addresse. Disse kaldes **icode** *(Instruktion kode)* og **ifun** *(instruktions funktion)*. En oversigt over disse funktionskoder kan ses i Figur 2.1. Generelt kan de siges at mængden af bytes der læses, skal adderes til *PC*.

Icode Icode gives ved `OP`, altså ved operationen, der udføres. Rådfør ved Figur 2.1.

Ifun Ifun gives ved funktionstypen der udføres. Igen kan der rådføres ved Figur 2.1.

Decode

Her indlæses de givne værdier for funktionen, i `valA` og eller `valB`. Dette gøres oftest fra `rA` og `rB`, men i nogle tilfælde også `%rsi`. For instruktioner noteret `valA` ← `R[...]` er værdien af `R[...]` nummeret på det givne register, der opereres på. Se Tabel 2.8.

Execute

`valE` er værdien, der gives efter *Execute* blokken er kørt. Her sættes konditions flagene `ZF`, `SF` og `OF`, hvis der skal tjekkes for henholdsvis 0 værdi, negativ værdi eller overflow. Samtidig noteres operationen som `valx OP valy`.

Memory

Her læses eller skrives til computerens hukommelse.

Writeback

Her skrives der tilbage, når funktionen er kørt.

PC update

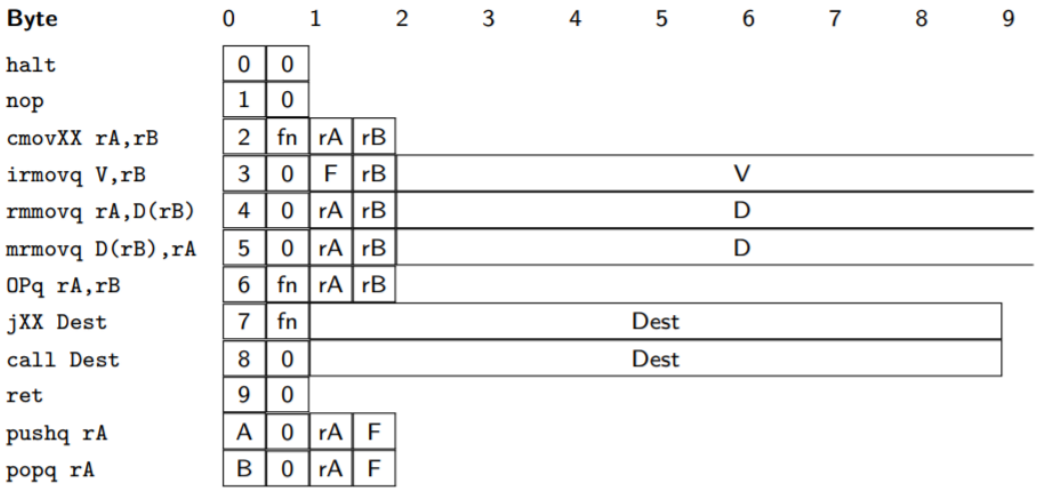
Program Counter opdateres. Her gives værdien af `valP`

2.2.2 Registre

Y86-Assembly har 16 registre til rådighed. Disse 16 registre vises i Tabel 2.8 Disse registre er næsten alle general purpose, dog med undtagelse af `%rsp`.

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	Intet register	F

Tabel 2.8: Oversigt over registre i Y86 Assembly



`rrmovq rA, rB`	2	0	Flyt fra register til register
`cmovle rA, rB`	2	1	Flyt hvis mindre eller lig
`cmovl rA, rB`	2	2	Flyt hvis mindre end
`cmove rA, rB`	2	3	Flyt hvis lig med
`cmovne rA, rB`	2	4	Flyt hvis ikke lig
`cmovge rA, rB`	2	5	Flyt hvis større eller lig
`cmovg rA, rB`	2	6	Flyt hvis større end
`addq rA, rB`	6	0	Addition
`subq rA, rB`	6	1	Sutraktion
`andq rA, rB`	6	2	Bitvis AND
`xorq rA, rB`	6	3	Bitvis XOR
`jmp Dest`	7	0	Ikke-konditionelt hop
`jle Dest`	7	1	Hop hvis mindre eller lig
`jl Dest`	7	2	Hop hvis mindre end
`je Dest`	7	3	Hop hvis lig med
`jne Dest`	7	4	Hop hvis ikke lig
`jge Dest`	7	5	Hop hvis større eller lig
`jg Dest`	7	6	Hop hvis større end

Figur 2.1: Liste over instruktion- og funktionskoder

Kapitel 3 Caching

3.1 Politik

Inden for caching findes der politiker, der definere hvor i cachen, ting skal gemmes. Der findes *Placeringspolitik* og *Erstatnignspolitik*. **Placeringspolitik** definerer hvor i cachen at en blok placeres, hvor **Erstatningspolitik** definerer, hvad der skal erstattes.

3.2 Hit og miss

Hit og miss forekommer, når der laves en forespørgsel til cachen. Et hit eksisterer, når der laves en forespørgsel til cachen, og den efterspurgte værdi findes i cachen. Et miss er modsat, når der laves en forespørgsel, og den efterspurgte værdi ikke eksisterer i cachen, hvorved den skal hentes i hukommelsen.

3.2.1 Kold miss

Et koldt miss, også kaldet et tvungent miss, opstår når cachen er tom. Dette betyder at der ikke er nogen værdier i cachen, og der derfor kun kan forekomme et miss.

3.2.2 Konflikt miss

Et konflikt forekommer når flere blokke spørger efter den samme plads, men cachen stadig har flere pladser tilgængelig. Dog har de fleste cache implementationer restriktioner, der er sat for at undgå denne type miss. Dette kan eksempelvis være at blok `B` skal placeres i `B mod 4` på niveau k.

3.2.3 Kapacitets miss

Dette forekommer når mængden af aktive caceh blokke (working sets) overgår cachens kapacitet.

3.3 Typer af cache

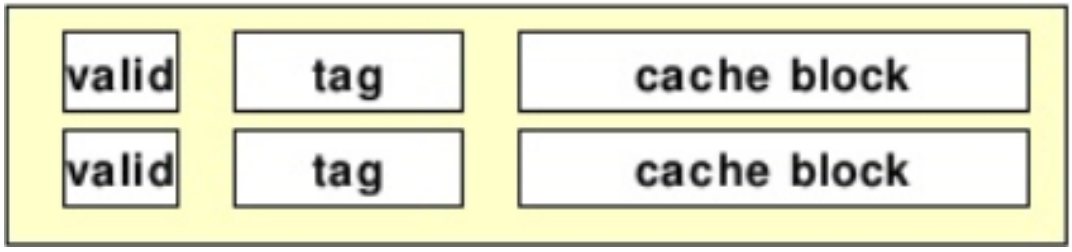
Der findes følgende typer af caching i hukommelses-hierakiet.

Cache type	Hvad caches	Hvor er cachen	Cycler	Styret af
Registre	4-8 bytes	CPU kerne	0	Compiler
TLB	Addresse overs.	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blokke	On-chip L1	4	Hardware
L2 cache	64-byte blokke	On-chip L2	10	Hardware
Virtuel hukommelse	4-KB sider	Primær huk.	100	Hardware + OS
Buffer cache	Dele af filer	Primær huk.	100	OS
Disk cache	Disk sektor	Disk controller	100000	Disk firmware
Netværks buffer cache	Dele af filer	Lokal disk	10000000	NFSC
Browser cache	Websider	Lokal disk	10000000	Webbrowser
Webcache	Websider	Servers	1000000000	Proxy server

Tabel 3.1: Caching i hukommelses-hierakiet

3.4 Cache opbygning

Cachen er opbygget af blokke, af størrelsen $S \cdot E$ elementer. En blok ser ud som vist i Figur 3.1 En sådan blok består af en *Valid* bit, der fortæller om blokken indeholder



Figur 3.1: Repræsentation af en cache blok

information, der kan bruges i den givne kontekst. Der findes også et antal bytes, der gemmer på den ønskede information. Yderligere indeholder den et markat *tag*, der bruges til at skældne mellem information i cache-linjen, fra hvor det starter i hukommelsen.

For hurtigt at kunne fremsøge information i cachen, opdeles addressen for det ord der ledes efter. Denne deles i *tag*, *index* og *offset*. De mindst betydende bits, bruges som offset, de midterste bits bruges til sæt indeks, og de mest betydende bits, bruges til tagget. For sæt indeks, gælder det at $2^s = S$, altså mængden af sæt i cachen. Offset vil for et 8-byte ord altid være 3, da offset er 0-indekseret, og derfor ikke må overskride antal gyldige bytes. Jvf. Tabel 1.1. Dette skyldes at man vil kunne addressere alle bytes i blokken. De resterende bits, bruges til at danne et tag, for at kunne skældne mellem sæt.

For at kunne tjekke sæt, tag og værdi, kan den givne addresse konverteres til binære tal, hvorved det binære tal kan opdeles i de respektive grupper. For et 2-byte ord `0xAC` (`1001 1011`), i en cache med 8 sæt, betyder det at strukturen vil se ud som

vist i Tabel 3.2. Med denne tabel i tankerne, kan ordet nu opdeles som `10.01 1.011`,

7	6	5	4	3	2	1	0
t	t	s	s	s	b	b	b

Tabel 3.2: Opdeling af 2-byte ord

hvorved tag, sæt, offset og værdi kan findes. **Bemærk** at offset er 0-indekseret.

3.5 Lokalitet

Om lokalitet kan følgende siges.

- Programmer der ofte tilgår de samme variabler har god temporal lokalitet.
- Programmer med *Stride-k* reference mønster. Jo mindre *Stride* jo bedre, hvor programmer med et *Stride-1* mønster, har den bedste spatialle lokalitet. Dette skyldes at programmet ikke skal hoppe i hukommelsen.
- Løkker har god temporal og spatial lokalitet, med respekt til operationerne og stride. Des mindre løkke-krop, des bedre lokalitet.

3.6 Caching

Kapitel 4 Pipeline

En pipeline er en sekventiel udførsel af en gruppe opgaver. Dette gør at opgavens *throughput*, altså hvor hurtigt opgaven løses, stiger. Dette kan dog også forhøje *latency*, hvorved opgaven gøres langsomere.

4.1 Throughput og Latency

I CART måles *Circuit delay* som picosekunder (ps). I et system, hvor der eksisterer 1 blok computationel logik, med et delay på 300ps, og et register load på 20ps, gives et throughput på 3.12GIPS. Dette kan udregnes på formlen:

$$Throughput = \frac{instruktion}{delay} \cdot \frac{1000ps}{1ns} \tag{4.1}$$

Throughput angives i GIPS *(Giga-instruktioner per sekund)*, og skal derfor ganges med 1000. Den fulde tid det tager at køre en beregning fra start til slut, kaldes latency, og er i ovenstående eksempel 320ps. Latency i *n-stage* kan udregnes ved følgende formel:

$$Latency = clock\ cycler \cdot instruktion\ delay \tag{4.2}$$

Med disse formler kan en forbedring og ny latency beregnes. Dette gøres ved brug af følgende to formler:

$$\Delta_{Throughput} = \frac{Throughput_{new}}{Throughput_{old}} \tag{4.3}$$

$$\Delta_{Latency} = \frac{Latency_{new}}{Latency_{old}} \tag{4.4}$$

Den øgede latency skyldes at der er tilføjet mere hardware, og flere pipeline registre. For bedre forståelse se lærebogen s. 451.

Unit	Seconds
1000ps	$1e^{-9}$
1ns	$1e^{-9}$

Tabel 4.1: Convertering af Picosekunder og nanosekunder

4.2 Data hazard

Data hazard forekommer, når instruktioner prøver at tilgå den samme data, under eksekvering. Dette betyder at en instruktion, skal have skrevet tilbage, før et givent register kan bruges igen.

Kapitel 5 Øvelser

Kapitel 6 Kursusgang 9

Dette afsnit vil behandle Kursusgang 9 i CART

6.1 Practice Problems

I denne sektion vil de givne Practice Problems for Kursusgangen blive gennemgået.

6.2 Practice Problem 6.5

Reads		Writes	
Sequential read throughput	550 MB/s	Sequential write throughput	470 MB/s
Random read throughput (IOPS)	89,000 IOPS	Random write throughput (IOPS)	74,000 IOPS
Random read throughput (MB/s)	365 MB/s	Random write throughput (MB/s)	303 MB/s
Avg. sequential read access time	50 μs	Avg. sequential write access time	60 μs

Figur 6.1: Performance characteristics of a commercial solid state disk. Source: Intel SSD 730 product specifictations. IOPS is I/O per seconds. throughput numbers are based on reads and writes of 4 KB blocks. (Intel SSD 730 product specifictations. Intel Cooperation)

As we have seen, a potential drawback of SSDs is that the underlying flash memory can wear out. For example, for the SSD in Figur 6.1, Intel guarantees about 128 petabytes ($128 \cdot 10^{15}$) of writes before the drive wears out. Given this assumption, estimate the lifetime (in years) of this SSD for the following workloads:

1. *Wors case for sequential writes:* The SSD is written to continuously at a rate of 470 MB/s (The average sequential write throughput of the device).

2. *Worst case for random writes:* The SSD is written to continuously at a rate of 303 MB/s (the average random write throughput of the device)

3. *Average case:* The SSD is written to at a rate of 20 GB/day (the average daily write rate assumed by some computer manufacturers in their mobile computer workload simulations)

6.2.1 Udregninger til 6.5

Det er givet at $1PB = 10^9 MB$. Samtidig vides det at der er 86400 sekunder på en dag.

1. Med denne information kan følgende formel bruges til at udregne levetiden for en SSD ved worst case load:

$$\left(10^9 \cdot 128\right) \cdot \left(\frac{1}{470}\right) \cdot \left(\frac{1}{(86400 \cdot 365)}\right) = \frac{800000}{92637} \approx 8.6359 \tag{6.1}$$

2. Samme formel kan bruges til at udregne worst case for random writes

$$\left(10^9 \cdot 128\right) \cdot \left(\frac{1}{303}\right) \cdot \left(\frac{1}{(86400 \cdot 365)}\right) = \frac{8000000}{597213} \approx 13.396 \tag{6.2}$$

3. Formlen kan også bruges til at udregne vores average case. Dog skal sekunder ikke bruges i dette tilfælde, da der arbejdes med en tidsfaktor af dage.

$$\left(10^9 \cdot 128\right) \cdot \left(\frac{1}{20000}\right) \cdot \left(\frac{1}{365}\right) = \frac{1280000}{73} \approx 17534.24658 \tag{6.3}$$

Udregningerne er i år.